Symbolische manipulatie

Rik Voorhaar (3888169) - Jan-Willem van Ittersum (3992942) - Jurre Corver (3905985)
Begeleider: Joost Houben

3 juli 2015

1 Introductie

In deze opdracht hebben we een eigen computeralgebrasysteem (CAS) ontwikkeld om symbolisch te kunnen rekenen zoals dat bijvoorbeeld in Mathematica gebeurt. Wiskundige formules worden hiervoor opgeslagen in een zogenaamde expressie-boom. Behalve dat deze representatie kan worden gebruikt om berekeningen te doen, is deze geschikt voor symbolische manipulaties, zoals optellen, vermenigvuldigen, maar ook differentiëren en oplossen van sommige polynoom vergelijkingen. De code behordende bij dit project kan gevonden worden op `https://github.com/JurreCorver/SymbolischeManipulatie`.

2 Theorie

Een expressie-boom is een samenhangende, gerichte graaf, waarbij elke knoop (op één knoop na) exact één inkomende zijde heeft en een willekeurig aantal uitgaande zijden. De unieke knoop zonder inkomende zijde noemen we *de wortel*, een knoop met minstens één uitgaande zijde noemen we *een interne knoop* en een knoop zonder uitgaande zijde noemen we *een blad*. De knopen waarop de uitgaande zijden van een interne knoop uitmonden, heten *de kinderen* van die interne knoop.

In de knopen van deze expressie-boom wordt een berekening opgeslagen. Een blad is een constante of een variabele, bijvoorbeeld 3 of x. Een interne knoop (de wortel is ook een interne knoop) is een operator of een functie. Naast de binaire operatoren optellen $+$, aftrekken $-$, vermenigvuldigen $*$, delen $/$, modulo $\%$ en machtsverheffing $**$ die gekenmerkt worden door het feit dat er zowel links als rechts van de operator één argument nodig is, is er een negatie operator $-$ die alleen een argument rechts heeft. Vergelijk bijvoorbeeld de binaire operator $-$ in $5-x$ met de negatie operator $-$ in $-x+2$. Een voorbeeld van een expressieboom zie je in figuur 1.

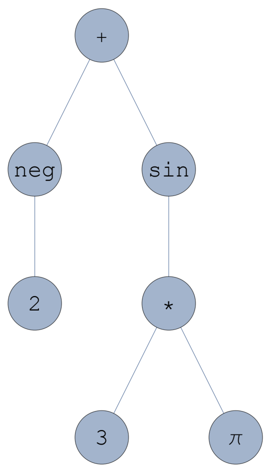

Figure 1: Voorbeeld van een expressieboom die de expressie $-2+\sin(3*\pi)$ voorstelt. In de zogenaamde postfix notatie, gedefinieerd in sectie 2.1, wordt deze expressie weergegeven als $2\,\mathrm{neg}\,3\,\pi*\sin+$. **JW: er gaat iets fout met Engels/Nederlands in de figuren, ze heten nu ‘figure’**

2.1 Infix en postfix notatie

De wiskundige notatie waaraan wij gewend zijn, waarbij operatoren tussen operanden worden geplaatst, noemen we ook wel de infix notatie. Een simpel voorbeeld hiervan is $1+2$ waarbij 1 en 2 de operanden zijn en $+$ de operator. Het nadeel van deze notatie is dat de volgorde waarin de operatoren moeten worden toegepast niet altijd direct duidelijk is, waardoor er haakjes gezet moeten worden. Een voorbeeld hiervan is $3+3/3$. Als delen eerst wordt uitgevoerd, is de uitkomst van deze expressie gelijk aan 3. Als optellen eerst wordt uitgevoerd, dan is de uitkomst gelijk aan 2.

Om dit probleem op te lossen kan de postfix ofwel reversed Polish notation worden ingevoerd. Hierbij volgt de operator direct op haar operanden en bestaat er geen onduidelijkheid over de volgorde van toepassing. Een voorbeeld hiervan is $3\ 3+3\ /$, dat eenduidig 2 als antwoord heeft. Ook voor functies kan deze notatie worden toegepast. Zo kunnen we de infix notatie gegeven door $f(1,2,3)$ in postfix notatie schrijven als $1\,2\,3\,f$. Wel is een nadeel hiervan dat het aantal argumenten van de functie van tevoren duidelijk moet zijn.

Gebruikers zullen hun invoer over het algemeen liever in de infix notatie doen. Daarom is er een algoritme nodig om de infix notatie om te schrijven naar postfix notatie en deze daarna in een expressieboom om te zetten. Hiervoor is het shunting-yard algortime geïmplementeerd.

3 Algoritmen

3.1 Shunting-yardalgoritme

Een gebruiker vindt infix notatie het makkelijkst te gebruiken, en zal graag zijn invoer dus ook in infix notatie aan een programma willen doorvoeren. De meest nuttige manier om een expressie op te slaan en te bewerken blijkt echter in de vorm van een expressieboom te zijn. Het Shunting-yardalgoritme maakt de vertaalslag van de infix notatie van de gebruiker naar postfix notatie, en de vertaling van postfix notatie naar een expressieboom is vrij gemakkelijk. Het algoritme is gepubliceerd door Dijkstra in 1961.

We gaan niet het algoritme in volledig detail beschrijven, we behandelen het slechts in grove lijnen aan de hand van voorbeelden. **JW: dat is niet wat je hier beneden doet: daar behandel je juist het algoritme in detail en geef je geen voorbeelden.** We beginnen met een string met een expressie in infix notatie, bijvoorbeeld `'2-3*3'`. Dit splitsen we dan op in kleine stukken genaamd 'tekens'. In dit geval: `'2'`,`'-'`,`'3'`,`'*'`.

Dan maken we vervolgens twee lijsten aan, de `stack` en de `output`. We gaan dan van links naar rechts door de lijst van tekens heen en volgen de volgende procedure uit:

1. Indien het teken een getal is, stop hem in de `output`.
2. Als het teken een komma is, verplaats dan steeds het bovenste element van de `stack` naar de `output` totdat het laatste element van de `stack` een `'('` is.
3. Als het teken een binaire operator `-`,`+`,`*`,`/`,`%` of `**` is, blijf dan aan de hand van de *prioriteit* van de operatoren van de `stack` naar de `output` verplaatsen totdat het laatste element van de `stack`

geen operator meer is. De prioriteit is een getal dat er voor zorgt dat de standaard volgorde van de operatoren (Meneer Van Dalen Wacht Op Antwoord) wordt uitgevoerd. De `-` teken wordt dan op een speciale manier behandeld, omdat het zowel negatie als aftrekken kan betekenen.

4. Als het teken `'('` is, kijk dan of het laatste element op de `output` een functie is. Zo ja, verplaats dan de functie naar de `stack`. Stop het haakje op de `output`.
5. Als het teken `')'` is, verplaats dan elementen van de `stack` naar de `output` totdat er een `'('` wordt tegengekomen. Verwijder dan de `'('` van de `stack`. Indien de bovenkant van de `stack` nu een functie is, voeg hem dan toe aan de output.
6. Als het teken een bekende variabele (e.g. `'pi'`) is, voeg zijn waarden dan toe aan de `output`.
7. Als het teken in geen van de bovenstaande gevallen zit, voeg hem dan als onbekende variabele toe aan de `output`.
8. Als we door de lijst van tekens heen zijn, voeg dan alle elementen van de `stack` toe aan de `output`.

Deze procedure geeft dan een uitdrukking in postfix notatie terug. Met het volgende algoritme wordt dan de postfix notatie naar een expressieboom veranderd. We itereren de volgende procedure over de `output` heen, en noemen de elementen van `output` weer tekens. We hebben om te beginnen eerst weer een lege `stack`.

1. Als het teken een functie is, haal dan n elementen van de `stack` af, waar n de hoeveelheid argumenten is die de functie aanneemt. Voeg vervolgens de functie als knoop toe de `stack` met als kinderen de n argumenten.
2. Als het teken een binaire operator is, voeg de operator als knoop toe aan de `stack` met als kinderen de twee operanden.
3. Als het teken geen functie of operator is, voeg hem dan aan de `stack` toe.

Na deze procedure hebben we bij invoer van geldige syntax slechts één element over in de `stack`. Dit element is dan de expressieboom met als bovenste knoop de wortel van de boom. Het gehele algoritme om infix notatie om te zetten in postfix notatie en vervolgens in een expressieboom is van orde $\mathcal{O}(n)$, aangezien er precies één keer over de lijst van tekens word geïtereerd, en ook precies één keer over de `output` wordt geïtereerd (en de output is vervolgens maximaal even lang als de lijst van tekens).

3.2 Simplify algoritme

JW: zullen we hier ook `evaluate` noemen? Die is nu nog nergens gedefinieerd. Voor vele andere functies in het programma is het erg nuttig om symbolische vergelijkingen te kunnen versimpelen. Denk hier bijvoorbeeld aan expressies als `x-x` naar `0` te versimpelen. Daarom is de functie `simplify(expression)` geïmplementeerd die de expressie `expression` zoveel mogelijk probeert te versimpelen. We zullen hier een summiere omschrijving geven over de werking van deze functie. De `simplify` functie roept de `simplifyStep` functie net zo vaak, met een maximum van 20 keer, aan tot dat de uitdrukking niet meer veranderd na verdere toepassing van `simplifyStep`. De `simplifyStep` functie gebruikt standaard rekenregels voor rëele getallen om polynomiale uitdrukkingen zo ver mogelijk te versimpelen. De functie `simplifyStep` bestaat dan ook uit het toepassen van een reeks functies die allemaal één keer door de boom itereren en een specifieke rekenregel gebruiken om de expressie te versimpelen. Elk zo'n versimpelende functie itereerd precies een keer over alle knopen, en zijn dus orde $\mathcal{O}(n)$. De functie `simplifyStep` roept echter zichzelf ook bij zijn kinderen aan, en daarom zijn `simplifyStep` en `simplify` orde $\mathcal{O}(n)$. Bij functies versimpeld de functie enkel de argumenten. Verder heeft `simplify` een optioneel argument die bepaald of bij elke toepassing van `simplifyStep` de expressie wordt geëxpandeerd.

3.3 Grafische omgeving

De grafische omgeving maakt gebruik van `tk` voor de weergave. Deze interpreteert dan de invoer van de gebruiker en voert deze uit door `expression_template.py` in te laden. De gebruiker heeft dan de optie om de uitvoer weer te geven met LaTeX. Dit werkt door de expressie om te vormen tot LaTeXcode. Dit omzetten naar LaTeXcode wordt recursief gedaan, en iedere soort knoop in de expressieboom heeft een methode om de LaTeXcode van zichzelf terug te geven. Deze LaTeXcode wordt vervolgens naar een `.tex` bestand geschreven en uitgevoerd door `latex` aan te roepen, wat dan weer een `.dvi` bestand als uitvoer geeft. Dit `.dvi` bestand wordt dan met behulp van *GhostScript* omgevormd naar een `.png` bestand die wordt weergegeven door `tk`.

3.4 Functies

JW: hoort deze sectie hier wel thuis? Je beschrijft hier geen algoritmes, ik vind dit eerder in de documentatie passen. Het programma heeft een implementatie van enkele standaard numerieke functies zoals `sin, cos` en `exp`. Deze functies zijn interne knopen in de expressieboom. Als methode kunnen deze functies dan een numerieke waarden teruggeven van de functie toegepast op zijn kinderen. Verder is het mogelijk om de functies te differentiëren naar een variabele. Ook is het ook voor de gebruiker mogelijk om tijdens het draaien van het programma classen aan te maken die functie knopen representeren. **JW: bedoel je in de vorige zin dat users zelf functies kunnen definiëren?** Deze knopen kunnen dan door de gebruiker weer worden gebruikt in expressies.

3.5 solvePolynomial algoritme

Het solvePolynomial algoritme neemt een polynomiale vergelijking en een variabele als argument en probeert dan een oplossing te vinden via de volgende stappen. **JW: Heb wat kleine dingen hieronder gewijzigd, kan je checken of het nog steeds klopt?**

1. Test of er wel een vergelijking is ingevoerd. Als dit zo is, haal dan de rechterkant van de vergelijking naar de linkerkant via gewoonlijke algebra. Is dit niet het geval, neem dan aan dat een nulpunt van de ingevoerde expressie moet worden bepaald.
2. Sla de linkerkant van de vergelijk op als een lijst met coëfficiënten voor de machten van de variabele.

3. Controleer wat de kleinste macht k in de vergelijking is. Als $k < 0$, vermenigvuldig dan met de variabele tot de macht $-k$. Is $k > 0$, deel dan door de variabele tot de macht k. Voeg in het tweede geval 0 toe als oplossing van de vergelijking.

4. Controleer nu de graad van de vergelijking. Is de graad 0, dan bestaat er geen oplossing. Is de graad 1, 2 of 3 geef dan de oplossing gebruikmakend van bekende algoritmes. In andere gevallen geeft het algoritme geen oplossing.

5. Return de oplossing. **JW: wat is de oplossing? Een lijst met alle nulpunten? Staan meervoudige nulpunten daar ook meervoudig in?**

JW: heb je enig idee wat de looptijd is van dit algoritme?

4 Documentatie

4.1 Installatie

Om dit programma te gebruiken zijn naast een werkende Python 3.4 distributie enkele Python packages nodig. De niet-standaard packages zijn: `numpy`, `pillow`, `scipy`, `tkinter`. Verder heeft de grafische gebruikersomgeving ondersteuning nodig voor het weergeven van de output met behulp van LaTeX. Hiervoor is een werkende LaTeX distributie vereist. Daarbij is het ook benodigd om *GhostScript* geïnstalleerd te hebben. Voor *Windows* gebruikers is het verder specifiek vereist om de 32-bit versie van *GhostScript* te gebruiken en het pad naar `gswin32c.exe` toe te voegen aan de `%path%` systeemvariabele.

4.2 Grafische omgeving

De grafische omgeving kan gebruikt worden door `tkmain.pyw` uit te voeren. Dit kan ook vanuit de command-line door middel van het commando`python tkmain.pyw`. De verschillende componenten van de omgeving zullen worden uitgelegd met referentie naar figuur 2.

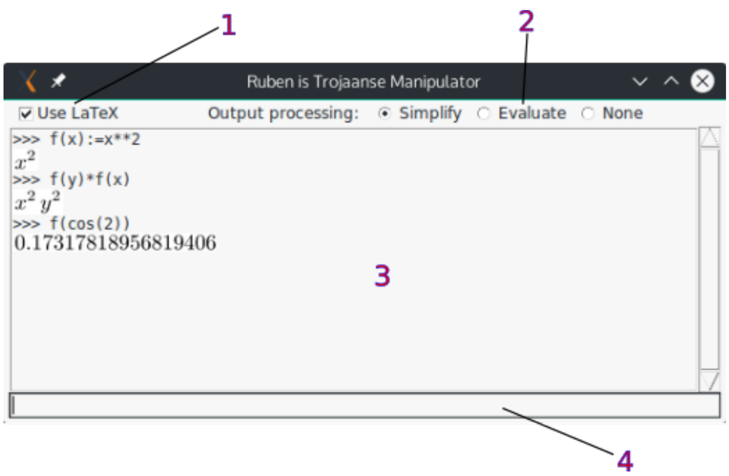

Figure 2: Screenshot van de grafische omgeving

1. Met de `Use LaTeX` wordt gespecificeerd of de output door LaTeXwordt verwerkt voor dat deze wordt weergegeven. Indien deze optie niet geselecteerd is zal de output als plain-text worden weergegeven.

2. De `Output processing` optie specificeerd hoe de input wordt verwerkt. Indien `Simplify` is aangevinkt zal de expressie zo veel mogelijk versimpeld worden. Bij het aanvinken van `Evaluate` wordt de expressie uitgerekend indien deze numeriek is, maar zal deze verder niet versimpeld worden weergegeven. Als laatste zorgt de `None` optie juist dat de expressie niet wordt berekend en direct wordt weergegeven.

3. Dit is het scherm waar de output wordt weergegeven. Hier wordt de input van de gebruiker met te tekens `>>>` ervoor weergegeven. Op de regel na de input wordt de bijbehorende output weergegeven. Indien er fouten optraden tijdens het uitvoeren van de input wordt dit hier ook weergegeven.

4. In dit scherm kan de gebruiker zijn input invoeren. Als de gebruiker de `Enter` toets indrukt wordt de input geëvaluaeerd. De pijltjestoetsen naar boven en onder kunnen worden gebruikt om de vorige input nog een keer te gebruiken.

4.3 Syntax

De invoer ondersteund standaard operaties op getallen die met symbolen `*, +, -, **, /, %` kunnen worden ingevoerd. Verder kunnen ronde haaken `( )` worden gebruikt. Enkele voorbeelden: **JW: Heb je nu simplify, evaluate of None aanstaan?**

```
>>> 2+2-1
1
>>> (2**3)/2
4
>>> 7 % 3
1
```

Ook zijn standaard functies als `sin, ln, gamma, gcd` geïmplementeerd. De lijst met alle geïmplementeerde functies is te vinden in de volgende sectie. Er is verder ook ondersteuning voor complexe getallen. De imaginaire constante wordt aangeduid met `i`. Andere standaard constanten zijn `pi, e, phi`. Voorbeeld:

```
>>> i**2
−1
>>> exp(i*pi)-1
0
```

Naast standaard operaties is het ook mogelijk om zelf functies en constanten toe te voegen. Hiervoor kan `:=` gebruikt worden. Als links van de `:=` alleen een symbool staat, dan wordt wat rechts van de `:=` staat opgeslagen onder naam van het symbool aan de linkerkant. We kunnen bijvoorbeeld het volgende doen:

```
>>> x:=5
5
>>> x**2
25
```

Functies kunnen op een vergelijkbare manier worden toegevoegd. Neem als voorbeeld:

```
>>> f(x,y):=sin(x)*cos(y)
```
$\sin(x)\cos(y)$

```
>>> f(pi/2,pi)
```
-1

Als laatste is er ondersteuning voor een gelijkheids operator met `==` als symbool. Deze kan bijvoorbeeld worden gebruikt bij invoeren van vergelijkingen. **JW: Wat moet je doen als je een of andere expressie hebt in 'x' en deze wil evalueren in, zeg, $x = 5$? Werkt expressie.evaluatedictionary?**

4.4 Lijst van commando's

Rik: na afloop op alfabetische volgorde zetten

`exit()`
: Sluit het programma af.

`d(f(x),x)`
: Berekent de afgeleide van `f(x)` naar de variabele `x`. Indien de afgeleide van de functie niet bekend is wordt een foutmelding gegeven. Voorbeeld:

```
>>> d(sin(x)+2,x)
```
$\cos(x)$

`sin(x), arcsin(x), cos(x), arccos(x), tan(x), arctan(x), ln(x), exp(x), floor(x), gamma(x)`
: Geeft de bijbehorende standaardfunctie als functie van `x`.

`log(x,y)`
: Geeft het logaritme van `x` in basis `y`. Equivalent aan `ln(x)/ln(y)`.

`polygamma(x,y)`
: Geeft de `y`'de orde polygamma functie van `x`.

Rik: misschien ook nog even round en ceil toevoegen voor compleetheid?

`solvePolynomial(vergelijking, var)`
: Geeft de oplossing van een polynomiale `vergelijking` in de variabele `var`. Bijvoorbeeld $x^{10} = 1024$ of $x^2 + 1/x = 10$. **JW: liever geef je een voorbeeld met wat de user moet invoeren en wat je dan als output krijgt**

`numIntegrate(expressie, x, l, r, numsteps)`
: Numeriek bepalen van $\int_{\mathtt{l}}^{\mathtt{r}}$ `expressie` `dx` door middel van een Riemann benadering in `numsteps` stappen.

`gcd(n,m)`
: Geeft de grootste gemene deler van twee gehele getallen `n` en `m`.

`polQuotient(pol1, pol2, x), polRemainder(pol1, pol2, x)`
: Geeft het quotient respectievelijk de rest van de deling van `pol1` door `pol2` met rest, waarbij `pol1` en `pol2` polynomen in `x` zijn.

`polIntQuotient(pol1, pol2, x), polIntRemainder(pol1, pol2, x)`
: Geeft het quotient respectievelijk de rest van de deling van het polynoom `pol1` door het polynoom `pol2` met rest *over de gehele getallen*, waarbij `pol1` en `pol2` polynomen met gehele coëfficiënten in `x` zijn.

`polContent(pol,x)`
Geeft de content van het polynoom `pol` in `x`.

`polGcd(pol1,pol2,x)`
Geeft de grootste gemene deler van twee polynomen `pol1` en `pol1` met gehele coëfficiënten in `x`.

5 Taakverdeling

In het verslag is de volgende taakverdeling gehanteerd:

`Jurre`
Outline van het verslag gemaakt, solvePolynomial aan het verslag toegevoegd en als commando **JW: wat heb je met solvePolynomial gedaan?**, theorie over postfix notatie geschreven, taakverdeling geschreven, bronvermeldingen toegevoegd.

`Rik`
Stukje theorie over shunting-yardalgoritme, grootste deel van de documentatie, stukje over user interface geschreven.

`Jan-Willem`
Introductie en theorie over expressie-bomen geschreven, polynoomfuncties toegevoegd aan documentatie, controle op spelling en stijl.

De volgende taakverdeling was van toepassing bij het schrijven van de software:

`Jurre`
solvePolynomial geschreven, aantal simpele nodes toegevoegd vergelijkbaar met MulNode,Simpele functies toegevoegd. Mindeg functie toegevoegd, Shunting-yard ietwat aangepast

`Rik`
Neg-, eq-, funcNode toegevoegd, User Interface toegevoegd, simplifier gebouwd, shunting-yardalgoritme uitgebreid, differentiëren ingevoerd, user variables en functions toegevoegd, opsporen van bugs in de gehele code.

`Jan-Willem`
Aantal simpele nodes toegevoegd vergelijkbaar met SubNode, de deg functie toegevoegd, het Shunting-yard algoritme aangepast, het bestand polynomials geschreven, numeriek integreren toegevoegd.

References

JW: references is geen Nederlands woord. Wikipedia - Shunting-yard algorithm: `https://en.wikipedia.org/wiki/Shunting-yard_algorithm`. - geraadpleegd op 28 juni 2015 Houben, Joost. Expressie-bomen en Symbolische Manipulatie: WISB256 – Programmeren in de Wiskunde.